Why are the tensor dimensions as given for convolutional neural networks? - TensorFlow

I am having a bit of trouble understanding the dimensions of the tensors used in the set up of convolutional neural networks using TensorFlow. For example, in this tutorial, the 28x28 MNIST images are represented like this:

```
import TensorFlow as tf

x = tf.placeholder(tf.float32, shape=[None, 784])
x_image = tf.reshape(x, [-1,28,28,1])
```

Assuming I have ten training images, the reshaping above makes my input `x_image` a collection of ten sub-collections of twenty-eight 28-dimensional column vectors.

It seems more natural to use

```
x_image_natural = tf.reshape(x, [-1,28,28])
```

instead, which would return ten 28x28 matrices.

Illustration:

```
a = np.array(range(8))
opt1 = a.reshape(-1,2,2,1)
opt2 = a.reshape(-1,2,2)
print opt1
print opt2

# opt1 - column vectors
>>[[[[0]
>>[1]]

>>[[2]
>>[3]]]


>>[[[4]
>>[5]]

>>[[6]
>>[7]]]]

# opt2 - matrices
>>[[[0 1]
>>[2 3]]

>>[[4 5]
>>[6 7]]]
```

In a similar vein, is there an intuitive way to understand why the convolutional layers have dimensions `(height_of_patch, width_of_patch, num_input_layers, num_output_layers)` ? The transpose, seems more intuitive, in that it is ultimately a collection of patch-sized matrices.

*** EDIT ***

I'm actually curious about *why* the dimensions of the tensors are ordered they way they are.

For the inputs, X, why don't we use

```
x_image = tf.reshape(x, [-1,i,28,28])
```

which would create batch_size, `i` -sized arrays of 28x28 matrices (where `i` is the number of input layers)?

Similarly, why aren't the weight tensors shaped like `(num_output_layers, num_input_layers, input_height, input_width)` (which again seems more intuitive in that it is a collection of 'patch matrices.')

tensorflow

edited Nov 22 '16 at 14:45

asked Nov 21 '16 at 23:05

Fortunato
152 6

2 Answers

The way that one layer of 2-D convolution works is by sliding a 2D window/filter/patch across the input to compute "feature maps". Put into the context of this MNIST dataset, the inputs are

grayscale images, hence they are in the dimension of [height, width, num_channels] ([28, 28, 1]). Assume you decide to use a 3x3 window/filter/patch, this determines the first two dimensions of the weights of this convolution layer (height_of_path=3, width_of_path=3). The reason of doing this sliding across height and width dimension is for sharing neurons and to preserve statistical invariance (a bird is still a bird no matter where it appears in the picture), additionally, it also brings some benefit in lowering computation. Each channel/depth is thought as carrying unique information (in the RGB channel case, R=255 and G=255 say completely different things) and we do not wanna share neurons across different depth/channels. Hence the third dimension of the weights of a convolution layer is identical as the inputs' depth dimension (num_input_layers=1 in the first convolution layer in the MNIST case). The last dimension of the weights of a convolution layer is a hyper-parameter that user gets to decide. This number determines how many feature maps are produced after this convolution layer. And the bigger the value, the higher the computation costs.

A quick summary. For any 2D convolution layer, assuming it receives input X with dimension of:

X - [batch_size, input_height, input_width, input_depth]

Then the weights w of this convolution layer would have a dimension of:

w - [filter_height, filter_width, input_depth, output_depth]

This convolution layer outputs a y in dimension of:

y - [batch_size, output_height, output_width, output_depth]

Typically ppl make filter_height=filter_width, and often set filter_height=3, 5, 7. output_depth is a hyperparameter that user gets to decide. The output_height and output_width are determined based on the input_height, input_weight, filter_height, filter_width, the sliding choice and padding choice, etc.

For more information, I'd encourage reading the Stanford CS231 notes on ConvNet, I personally find it very clearly and insightfully explained.

Edit: The order of the dimension

As far as the order of the dimension goes, to my knowledge, it's more of a convention instead of "right" or "wrong". For one sample input, I think it's intuitive to order its dimension in the order of [height, width, channels/depth]. As a matter of fact, you can simply stick a sample matrix with this order of dimension into `import matplotlib.pyplot as plt; plt.imhow(sample_matrix)` to plot a human-eye-friendly image. I think the first three weight dimension order follows the conventional order of [height, width, depth]. I speculate that this consistency makes it easy to perform the convolution operation, as I read that one of the common implementation of this step is to flatten the 3D tensor into 2D and use matrix multiplication libraries underneath. I imagine you can change the order of the dimension into the way you want it to be as long as the actual computation btw dimensions are done correctly.

edited Nov 22 '16 at 19:41

answered Nov 22 '16 at 0:58

Zhongyu Kuang
1,493 1 12 25

I don't believe that the order of dimensions is arbitrary, as the pixels have a set spatial relation to each other. If we let `a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])` , there is a definite difference between using `row_col_depth = reshape([-1,2,2,4])` and `depth_row_col = reshape([-1,4,2,2])` in terms of what each of the 4 sub-matrices created (which can by printing each collection along the `depth` axis.) I think the answer has more to do with the second part of your edit: collapsing the 3D tensors into 2D. The link you shared gives a pretty solid explanation – Fortunato Nov 22 '16 at 20:13

I agree that the order of dimensions is not arbitrary, hence I said I believe it's more a convention. Conventionally when we perceive digital images, the last dimension maps to depth/channels. I think ppl can do the order any (unconventional) way they want as long as they make sure the math is done right but they'd write their own library to perform the computation, b/z after all the computer doesn't even see a 2D matrix in 2-D, it sees as a long series of numbers. And the nature of `reshape` is just rolling around the index of different dimensions. – Zhongyu Kuang Nov 22 '16 at 20:33

Ah, ok. I see your point. In that case, I agree – Fortunato Nov 22 '16 at 22:02

I believe the extra dimension of 1 in the shape is for the channel, which is required for `conv2d` . In other words, if the MNIST images were in color it would be 3 (for RGB), but since they are in grayscale it's just 1.

I don't have an intuitive explanation for the dimension order -- maybe someone else will.

answered Nov 21 '16 at 23:58

Neal

342 2 12